

# OMP2HMPP: HMPP Source Code Generation from Programs with Pragma Extensions

Albert Saà-Garriga  
Universitat Autònoma de  
Barcelona  
Edifici Q, Campus de la UAB  
Bellaterra, Spain  
albert.saa@uab.cat

David Castells-Rufas  
Universitat Autònoma de  
Barcelona  
Edifici Q, Campus de la UAB  
Bellaterra, Spain  
david.castells@uab.cat

Jordi Carrabina  
Universitat Autònoma de  
Barcelona  
Edifici Q, Campus de la UAB  
Bellaterra, Spain  
jordi.carrabina@uab.cat

## ABSTRACT

High-performance computing are based more and more in heterogeneous architectures and GPGPUs have become one of the main integrated blocks in these, as the recently emerged Mali GPU in embedded systems or the NVIDIA GPUs in HPC servers. In both GPGPUs, programming could become a hurdle that can limit their adoption, since the programmer has to learn the hardware capabilities and the language to work with these.

We present OMP2HMPP, a tool that, automatically translates a high-level C source code (OpenMP) code into HMPP. The generated version rarely will differs from a hand-coded HMPP version, and will provide an important speedup, near 113×, that could be later improved by hand-coded CUDA. The generated code could be transported either to HPC servers and to embedded GPUs, due to the commonalities between them.

## Categories and Subject Descriptors

D.3.2 [Language Classifications]: Concurrent, distributed, and parallel languages; D.3.4 [Processors]: Translator writing systems and compiler generators

## General Terms

Parallel Computing

## Keywords

Source to Source Compiler, GPGPUS, HMPP, Embedded GPUs, parallel computing, program understanding, compiler Optimizations

## 1. INTRODUCTION

GPGPUs are potentially useful for speed up applications and are one of the main integrated blocks on heterogeneous platforms, an example of these could be the recently emerged Mali GPU which is used embedded systems or the more

studied NVIDIA GPUs, which are present on all of the top ten server of the November 2013 list [22]. Although Compute Unified Device Architecture (CUDA) [16] programming model from NVIDIA, RapidMind [21], PeakStream [18] and CTM [7] have made the use of GPUs, for general propose programming easier and efficient. Some graphically rich applications are initially developed for HPC servers and later ported to embedded or mobile platforms. The reduction in available resources means that the application is unlikely to work at the same performance level as it does on the desktop platform. However, due to the commonalities between them a code generated to work with HPC servers will be portable to embedded system. Nevertheless, in both cases the implementation of a program that has to work with GPGPUs will be complex and error-prone due to the programming complexity and the language paradigms. The proposed tool (OMP2HMPP), simplify this task freeing the programmer to learn a new language and to implement the program to work with GPUs.

High-performance computing (HPC) community has been alive for a long time, usually working with a couple of standards: MPI and, the chosen to be the source code input for our tool, OpenMP. OMP2HMPP use of OpenMP directive-based code as input since that facilitates the understanding of the source code blocks that can be paralyzed. These blocks will be transformed to work in GPUs using HMPP [6, 9] set of directives, which are meta-information added in the application source code that do not change the semantic of the original code and blinds the complexity of final architecture to the user. They address the remote execution (RPC) of functions or regions of code on GPUs and many-core accelerators as well as the transfer of data to and from the target. OMP2HMPP uses these directives to define and call GPU kernels, and minimize the use of the directives related to data transfers between CPU and GPU. OMP2HMPP is able to analyze the OpenMP blocks before to transform these and to determine the scope and the aliveness of each of variables used in that block. Finally, OMP2HMPP offers to the programmer a translated version of the original input code which could be able to work in GPUs.

OMP2HMPP is Source to Source compiler (S2S) based on BSC's Mercurium framework [15]. Mercurium [5] is a source-to-source compilation infrastructure aimed at fast prototyping and supports C and C++ languages and is mainly used in Nanos environment to implement OpenMP but since it

is quite extensible it has been used to implement other programming models or compiler transformations. This framework is used in order to implement our S2S transformation phases, providing us with the Abstract Syntax Tree (AST) as an easy access to the table of symbols. This information is analyzed through OMP2HMPP tool to parse and translate the original problem to an optimum version of HMPP.

## 1.1 HMPP Directives

The proposed tool is able to use the combination of the following HMPP directives:

- **Callsite:** Specifies the use of a codelet at a given point in the program. Related data transfers and synchronization points that are inserted elsewhere in the application have to use the same label.
- **Codelet:** Specifies that a version of the function following must be optimized for a given hardware.
- **Group:** Allows the declaration of a group of codelets.
- **Advanced Load:** Upload data before the execution of the codelet.
- **Delegate Store:** The opposite of the advancedload directive in the sense that it downloads output data from the HWA to the host.
- **Synchronize:** Specifies to wait until the completion of an asynchronous callsite execution.
- **Release:** Specifies when to release the HWA for a group or a stand-alone codelet .
- **No Update:** This property specifies that the data is already available on the HWA and so that no transfer is needed. When this property is set, no transfer is done on the considered argument.
- **Target:** Specifies one or more targets for which the codelet must be generated. It means that according to the target specified, if the corresponding hardware is available AND the codelet implementation for this hardware is also available, this one will be executed. Otherwise, the next target specified in the list will be tried. OMP2HMPP always use CUDA since we will test it in a server without OpenCL support.

With these directives OMP2HMPP is able to create a version that in the most of the cases will be equal to a HMPP hand-coded version of the original problem.

## 1.2 Related Work

In the recent years, many source-to-source compiler alternatives to the dominant GPU programming models (CUDA [16] and OpenCL [11]) have been proposed to overcome the GP-GPU programming complexity. The more similar alternatives to the tool proposed in this paper are presented to do a brief comparative of these. In contrast to OMP2HMPP, the following explained methods, would obtain a direct transformation to CUDA language, not to HMPP, which means that the CUDA programming complexity is directly exposed to the final user.

In one hand there are proposals that extend in one way or another current programming standards such as C/C++, OpenMP, etc. to trivialize this task. In the other hand, there are proposals that does not need any previous language extensions to transform the code to work and transform the code directly from CPU to GPUs.

One of the examples that includes language extensions is proposed in [17]. CAPS, CRAY, NVIDIA and PGI, which are members of the OpenMP Language Committee published OpenACC in November 2011, an standard for this directive-based programming that contribute to the specification of OpenMP for accelerators. In the same way, but not with the same consensus that OpenACC, in [13], a programming interface called OpenMPC is presented. This paper shows and extensive analysis of the actual state of the art in OpenMP to CUDA source-to-source compilers and CUDA optimizers. OpenMPC provides an abstraction of the complex of CUDA programming model and develops an automatic with a user-assisted tuning system. However, OpenMPC and OpenACC require time to understand the new proposed directives, and to manually optimize the data transfer between GPU and GPU. In contradistinction of both cases, OMP2HMPP just add one new OpenMP directive and the programmer forgets to deal with new languages and the optimization of these. Another option is hiCUDA directive-based language [10], which is a set of directives for CUDA computation and data attributes in a sequential program. Nevertheless, hiCUDA has the same programming paradigm than CUDA; even though it hides the CUDA language syntax, the complexity of the CUDA programming and memory model is directly exposed to programmers. Moreover, in contrast to OMP2HMPP, hiCUDA does not provide any transfer optimization. Finally [14] and [3], propose an OpenMP compiler for hybrid CPU/GPU computing architecture. In this papers they propose to add a directive to OpenMP in order to choose where the OpenMP block must be executed (CPU/GPU). The process is full blinded to the programmer and is a direct translation to CUDA. The main differences with OMP2HMPP does not provide any transfer optimization.

There are less proposals that tries to do direct transformation from C/C++ to CUDA without need any new language extension to transform the code. One of this cases is Par4All [1]. This tool is able to transform codes originally wrote in C or Fortran to OpenMP, CUDA or OpenCL. Par4All transform C/C++ source code and add OpenMP directives where the program thinks that can be useful. This transformation allows the transformation of the created OpenMP blocks to GPGPUs kernels by transforming the OpenMP directives to CUDA language. However, the transformation that this program done for CUDA have not take in account the kernel data-flow context and this is not optimum in data-transferences.

For source-to-source compiler infrastructure, there are many possible solutions as LLVM [12], PIPS [2], Cetus [8], ROSE [20] and the used Mercurium [5].

## 2. OMP2HMPP COMPILER

The main objective of OMP2HMPP is to transform OpenMP blocks into HMPP kernels and their calls. In order to de-

terminate the OpenMP blocks that have to be transformed OMP2HMPP use the directives proposed in [4], as is illustrated in Figure 1. For each block, OMP2HMPP made an outline to create HMPP codelets (GPU kernels) and, at the same time, extract information from the transformed code to determine if the parameters of the created function are used just as input or are updated inside this and therefore, are output. We shown an example of that kernel creation in Table 2 (line 19) for the OpenMP block shown in Table 1 (lines 24-30).

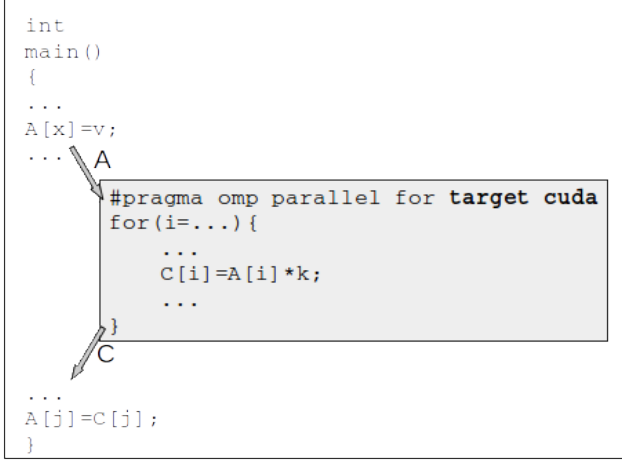


Figure 1: Context Example

After that transformation, OMP2HMPP analyze the context where the codelet is invoked, and do an accurate contextual analysis of the AST for each of the variables needed in the created codelet, to reduce the data transfers between CPU and GPU. OMP2HMPP is able to determine where is computed(CPU/GPU) the next or last access(read/write). OMP2HMPP search all the assignment (=) or assignment operation expressions (+, -, /, \*, =) and divide them in two expressions that contain the variables as operands.

OMP2HMPP use the aforementioned information to choose the best use of HMPP directives that minimize the number of data transfers. For a simple problem, as the proposed in Figure 1, where there is a OpenMP block to transform, OMP2HMPP will analyze the dependences of all the variables inside the kernel ( $A$  and  $C$ ) and process the extracted information. In Figure 1, OMP2HMPP has two variables to analyze  $A$  and  $C$ . In the case of  $A$ ,  $A$  has to be uploaded to GPU, but is not necessary to download after the kernel call because there is no read of that variable before the code finish. In contradistinction, in the case of variable  $C$ ,  $C$  has to be downloaded from GPU to CPU, but there is no need to upload that to GPU since the kernel do not do a read of  $C$  inside. With that information, OMP2HMPP will use an *advancedload* in the case of  $A$  and will put that directive as close as possible to the last write expression, to optimize the data transfer and improve the performance of the generated code as is shown in Figure 4b. In the case of  $C$ , OMP2HMPP will put a *delegatestore* directive, as far as possible of the kernel call, and that will increase the performance of the generated code, as is shown in Figure 5b.

Figures 4a and 5a illustrate the use of a bad transfer policy in the same problems.

Moreover, OMP2HMPP can deal in context situations in which the source code contains nested loops. OMP2HMPP determine if an operation over a variable is made inside a loop and adapt the data transfer to the proper context situation. We illustrate an example of a possible context situations in Figures 2 and 3. In the first figure, when OMP2HMPP wants to compute the loop in GPU, has to load before the values of variable  $A$ , which is needed to calculate the value of  $C$ . Since the last write in CPU of  $A$  is inside a loop with a different nested level than the GPU block, OMP2HMPP has to backtrack the nesting of loops in which is located the last write of  $A$ , to find the block shared by both loops. Then, as in the problem shown in Figure 1, OMP2HMPP optimize the load of  $A$  putting the *advancedload* directive as close as possible after the loop finish. We could change the same problem changing the block that is computed in GPU, as is shown in Figure 3. In this figure, the result of the GPU kernel is needed in CPU to compute  $C$ , but is not in the same loop level. In that case, the optimum way to put the *delegatestore* directive, will be just before the start of the nested loops where the computation of  $C$  is located.

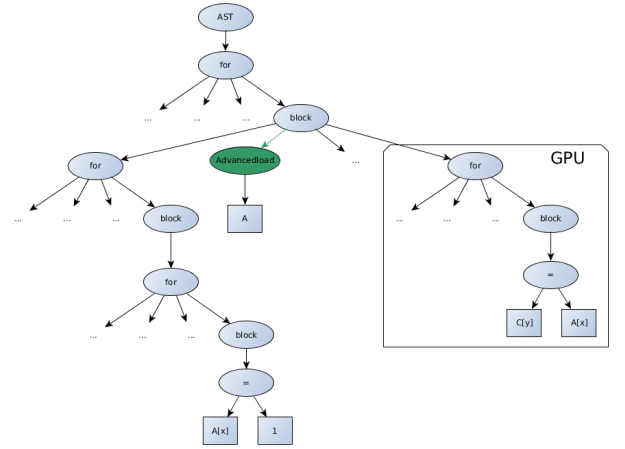


Figure 2: Data transfer in Loops Example

### 3. RESULTS

We shown in Table 2 an example of the resulting code for the problem illustrated in Table 1(3MM). In these figures there is exemplified a real case of use of the OMP2HMPP(explained in Section 2), we show that OMP2HMPP create a callsite and codelet for each OpenMP block, outlining the OpenMP block into a GPU kernel function, with the syntax requirements of HMPP language. At the same time, OMP2HMPP create a group to share variables between the created kernels using the combination of *group*, *mapbyname* and *noupdate* directives to avoid the unnecessary upload/download(marked in blue in Table 2). Finally, OMP2HMPP uses asynchronous GPU process of the first kernels since the calculated variables are not needed until the last kernel is invoked.

Furthermore, Table 2 also show the use of the context information and how OMP2HMPP deals with it, as was explained in Section 2. OMP2HMPP determine the kind of

```

1  #pragma hmpp <group1.0> _instr_for0.ol_28_main codelet, args[A, B, E].io=in
2  void _instr_for1.ol_28_main(int i, int ni, int j, int nj, double E[4000][4000], int k,
3      int nk, double A[4000][4000], double B[4000][4000])
4  {
5      for (i = 0; i < ni; i++)
6          for (j = 0; j < nj; j++)
7          {
8              E[i][j] = 0;
9              for (k = 0; k < nk; k++)
10                 E[i][j] += A[i][k] * B[k][j];
11          }
12  }
13 #pragma hmpp <group1.1> _instr_for1.ol_30_main codelet, args[F, C, D].io=in
14 void _instr_for1.ol_30_main(int i, int nj, int j, int nl, double F[4000][4000], int k,
15     int nm, double C[4000][4000], double D[4000][4000])
16 {
17     ...
18 }
19 #pragma hmpp <group1.1> _instr_for1.ol_32_main codelet, args[G].io=inout, args[E, F].io=in
20 void _instr_for1.ol_32_main(int i, int ni, int j, int nl, double G[4000][4000], int k,
21     int nj, double E[4000][4000], double F[4000][4000])
22 {
23     ...
24 }
25 int main(int argc, char **argv)
26 {
27     #pragma hmpp <group1.0> group, target=CUDA
28     #pragma hmpp <group1.0> mapbyname, E, A, B, F, C, D, G
29     int i, j, k;
30     int ni = 4000;
31     int nj = 4000;
32     int nk = 4000;
33     int nl = 4000;
34     int nm = 4000;
35     for (i = 0; i < ni; i++)
36         for (j = 0; j < nk; j++) {
37             A[i][j] = ((double) i * j) / ni;
38         }
39     #pragma hmpp <group1.0> _instr_for0.ol_28_main advancedload, args[A]
40     for (i = 0; i < nk; i++)
41         for (j = 0; j < nj; j++)
42             B[i][j] = ((double) i * (j + 1)) / nj;
43     #pragma hmpp <group1.0> _instr_for0.ol_28_main advancedload, args[B]
44     for (i = 0; i < nj; i++)
45         for (j = 0; j < nm; j++)
46             C[i][j] = ((double) i * (j + 3)) / nl;
47     #pragma hmpp <group1.0> _instr_for0.ol_30_main advancedload, args[C]
48     for (i = 0; i < nm; i++)
49         for (j = 0; j < nl; j++)
50             D[i][j] = ((double) i * (j + 2)) / nk;
51     #pragma hmpp <group1.0> _instr_for0.ol_30_main advancedload, args[D]
52     {
53         #pragma hmpp <group1.0> _instr_for0.ol_28_main callsite, args[E, A, B].noupdate=true, asynchronous
54         _instr_for0.ol_28_main(i, ni, j, nj, E, k, nk, A, B);
55         #pragma hmpp <group1.0> _instr_for0.ol_30_main callsite, args[F, C, D].noupdate=true, asynchronous
56         _instr_for0.ol_30_main(i, nj, j, nl, F, k, nm, C, D);
57         #pragma hmpp <group1.0> _instr_for0.ol_28_main synchronize
58         #pragma hmpp <group1.0> _instr_for0.ol_30_main synchronize
59         #pragma hmpp <group1.0> _instr_for0.ol_32_main callsite, args[G, E, F].noupdate=true, asynchronous
60         _instr_for0.ol_32_main(i, ni, j, nl, G, k, nj, E, F);
61     }
62     #pragma hmpp <group1.0> _instr_for0.ol_32_main synchronize
63     #pragma hmpp <group1.1> release
64     return 0;
65 }

```

Table 2: OMP2HMPP Generated Code Example

access of each variable needed in the created kernel, as is illustrated in the transformation of the first OpenMP block in Table 1, where the variable  $E$  has not been wrote before the block and therefore a load is not necessary. OMP2HMPP also analyze the host where a variable is used (CPU/GPU) as the shown in Table 1 with the variables needed for kernel `_instr_for0.ol_32_main`. These variables are all contained and updated in GPU and OMP2HMPP avoids to reload them. Finally, we exemplify the understand of loop context situation and that OMP2HMPP can identify in which loop is the use of an analyzed variable in the case of variable  $A$  (line 30 in Table 2), which is needed in HMPP kernel `_instr_for0.ol_28_main` (line 53 in Table 2) and there is no CPU write instructions for this variable between these lines. For that reason, OMP2HMPP put `advancedload` instruction after the last variable  $A$  assignment, but since this assignment is made inside a loop, OMP2HMPP postpone the load after the loop finish and add `advancedload` instruction in line 39.

To have an performance of the codes generated by OMP2HMPP, we created a set of codes extracted from the Polybench [19] benchmark and then, we compare the execution of these with the original OpenMP version, with a hand-coded CUDA version and with a sequential version of the same problem. In Figures 6a 6b, and 6c we show for each problem the speed-up comparison for the architectures showed in Table 3. In these figures, we show that OMP2HMPP made a good transformation for the originally OpenMP code, and obtain an average speedup of 113×. This speedup is less that the obtained in the execution of CUDA hand-coded code of the same problem in the most of the problems with an average speedup of 1.7×, but is quite similar in the covariance problem. Moreover, the average speedup obtained when we compare the generated code to the original OpenMP version is over 31×, which is a great gain in performance for an programmer that do not need any knowledge in GPGPU

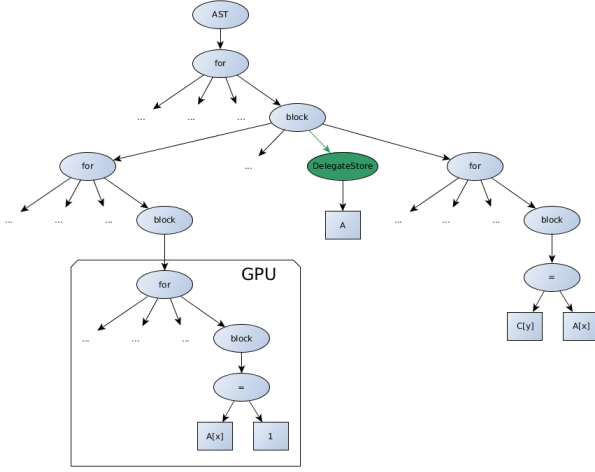


Figure 3: Data transfer in Loops Example

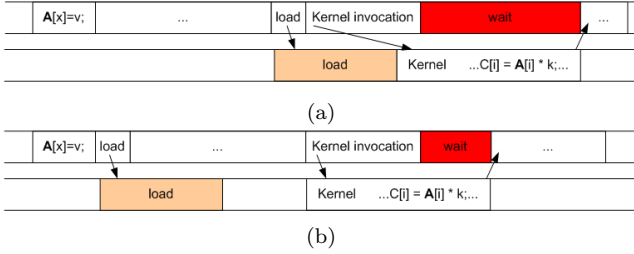


Figure 4: Advanced Load Directive Optimization. a) Variables are loaded when kernel is invoked. b) Variables are loaded as near as possible of the last CPU write.

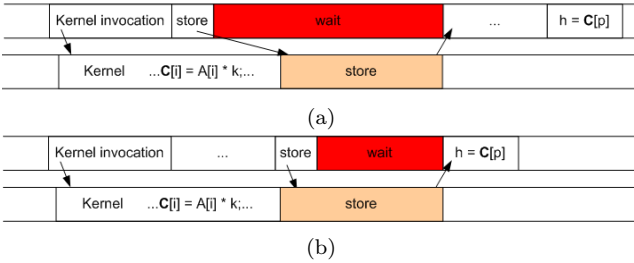


Figure 5: Delegate Store Directive Optimization. a) Variables are downloaded when kernel finish b) Variables are download as far as possible of the kernel finish, next to the first CPU read.

programming.

## 4. CONCLUSIONS

The programmer can use OMP2HMPP source to source compiler and avoid to expend time learning the meaning of HMPP directives or another GPUs language. The tested problems from Polybench benchmark obtains an average speedup of 113 $\times$  compared to the sequential version and an average speedup over 31 $\times$  compared to the OpenMP version, since OMP2HMPP gives a solution that rarely differ from the best HMPP hand-coded version. The generated version could be useful for more experimented users that

```

1  int main(int argc, char** argv) {
2      int i, j, k;
3      int ni = NI;
4      int nj = NJ;
5      int nk = NK;
6      int nl = NL;
7      int nm = NM;
8      for (i = 0; i < ni; i++)
9          for (j = 0; j < nk; j++) {
10             A[i][j] = ((double) i * j) / ni;
11         }
12     for (i = 0; i < nk; i++)
13         for (j = 0; j < nj; j++) {
14             B[i][j] = ((double) i * (j + 1)) / nj;
15         }
16     for (i = 0; i < nj; i++)
17         for (j = 0; j < nm; j++) {
18             C[i][j] = ((double) i * (j + 3)) / nl;
19         }
20     for (i = 0; i < nm; i++)
21         for (j = 0; j < nl; j++) {
22             D[i][j] = ((double) i * (j + 2)) / nk;
23         }
24     #pragma omp parallel private (j, k)
25     {
26         /* E := A*B */
27         #pragma omp for
28         for (i = 0; i < ni; i++)
29             for (j = 0; j < nj; j++) {
30                 E[i][j] = 0;
31                 for (k = 0; k < nk; ++k)
32                     E[i][j] += A[i][k] * B[k][j];
33             }
34         /* F := C*D */
35         #pragma omp for
36         for (i = 0; i < nj; i++)
37             for (j = 0; j < nl; j++) {
38                 F[i][j] = 0;
39                 for (k = 0; k < nm; ++k)
40                     F[i][j] += C[i][k] * D[k][j];
41             }
42         /* G := E*F */
43         #pragma omp for
44         for (i = 0; i < ni; i++)
45             for (j = 0; j < nl; j++) {
46                 G[i][j] = 0;
47                 for (k = 0; k < nj; ++k)
48                     G[i][j] += E[i][k] * F[k][j];
49             }
50     }
51     return 0;
52 }

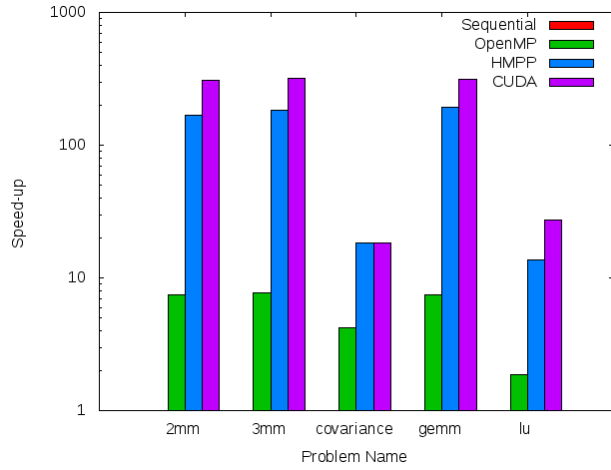
```

Table 1: OMP2HMPP Original Code Example

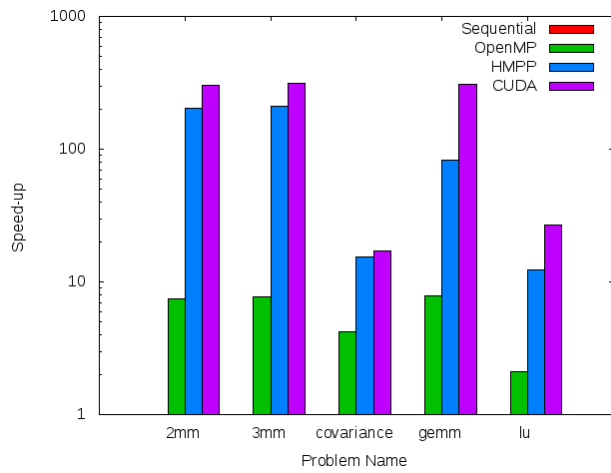
	B505 blade(1)	B505 blade(2)	B515 blade
Num. Processors	2	2	2
Processor	E5640	E5640	E5-2400
Memory	24 Gb	24 Gb	192Gb
GPU	Nvidia M2050	Nvidia Tesla C2075	Nvidia Tesla K20

want to have, without effort, a GPGPU code that will define an optimization starting point for their problems and then, implement a code that could get, for the tested problems, an speedup near 1.7 $\times$  compared with the version generated by OMP2HMPP, as is showed in Section 3.

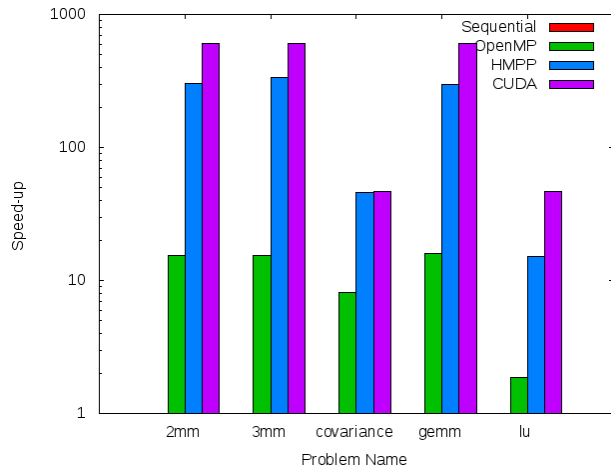
In future versions of the OMP2HMPP tool, it could be interesting to study the use of Par4All tool to have a complete automatic transformation from a sequential C/C++ source code to an HMPP parallelized version, taking the output OpenMP transformed version of this program as input of our proposal. Moreover, since OMP2HMPP is thought to work in heterogeneous architectures will be interesting to do an exploration of the possibilities to combine CPU and GPGPUs in the same problem. Nevertheless, if we want to use the generated code in in mobile or embedded systems it is important to be aware of power consumption in future



(a) B505 blade(1)



(b) B505 blade(2)



(c) B515 Blade

Figure 6: Speedup compared with the sequential version

versions, and to study the energy/time trade-off of the generated version.

## 5. ACKNOWLEDGMENTS

This work was partly supported by the European cooperative ITEA2 projects 09011 H4H and 10021 MANY, the CATRENE project CA112 HARP, the Spanish Ministerio de Economía y Competitividad project IPT-2012-0847-430000, the Spanish Ministerio de Industria, Turismo y Comercio projects and TSI-020100-2010-1036, TSI-020400-2010-120. The authors thank BULL SAS and CAPS Entreprise for their support.

## 6. REFERENCES

- [1] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon, et al. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.
- [2] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, P. Jouvelot, and R. Keryell. Pips: A framework for building interprocedural compilers, parallelizers and optimizers. Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris, 1996.
- [3] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. Igual, D. Jiménez-González, J. Labarta, et al. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.
- [4] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, et al. A proposal to extend the openmp tasking model for heterogeneous architectures. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 154–167. Springer, 2009.
- [5] J. Balart, A. Duran, M. González, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 8, 2004.
- [6] CAPS. Openhmpp directives, Mar. 2007.
- [7] A. CTM. Technical reference manual, Mar. 2006.
- [8] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.
- [9] R. Dolbeau, S. Bihan, and F. Bodin. Hmpps: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.
- [10] T. D. Han and T. S. Abdelrahman. hi cuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.
- [11] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
- [12] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

- [13] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.
- [14] H.-F. Li, T.-Y. Liang, and J.-L. Jiang. An openmp compiler for hybrid cpu/gpu computing architecture. In *Intelligent Networking and Collaborative Systems (INCoS), 2011 Third International Conference on*, pages 209–216. IEEE, 2011.
- [15] Nanos. Mercurium, Mar. 2004.
- [16] NVIDIA. Cuda sdk, Mar. 2007.
- [17] OpenACC Working Group. The OpenACC Application Programming Interface, Version 1.0. November 2011.
- [18] PeakStream. home page, Mar. 2006.
- [19] L.-N. Pouchet. Polybench: The polyhedral benchmark suite, 2012.
- [20] D. J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.
- [21] Rapid. home page, Mar. 2009.
- [22] TOP500.org. Top 500 supercomputer sites, Nov. 2013.